Bachelor Thesis

# Satogaeri

Markus Leitner (1117367)

`csap3977@student.uibk.ac.at`

17 December 2014

**Supervisor:** Univ.-Prof. Dr. Aart Middeldorp

# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

_____                    _____
          Datum                                      Unterschrift

**Abstract**

# Contents

# 1. Introduction

This thesis discusses the bachelor project 'Satogaeri' for the computational logic research group at the University of Innsbruck. The project was proposed as part of the logic puzzle focus[1] of the mentioned research group. Over the years several solvers for Japanese puzzles similar to Sudoku have been developed to demonstrate how logic can be used in problem domains known by a broad audience.

The project had the following aims:

1. implement a solver for Satogaeri puzzles,

2. implement a generator for Satogaeri puzzles,

3. implement a tool that has (at least) the same functionality as the Satogaeri player of Nikoli[2] and incorporates (1) and (2).

While reading this thesis one will be lead through the steps how those aims were met and additionally will be given some further background information concerning Satisfiability Modulo Theories (Section 3) and JavaFX (Section 7).

However everything can be improved therefore some thoughts about future optimisations and work are added in the Conclusion (Section 9).

Furthermore there was an optional task to prove that the Satogaeri puzzle problem is an NP-complete problem. Unfortunately this point was not fulfilled during the project.

---

[1]http://cl-informatik.uibk.ac.at/software/puzzles/
[2]http://www.nikoli.com/en/puzzles/satogaeri/

# 2. Satogaeri

Satogaeri is a logic puzzle published by the Japanese company Nikoli Ltd. in 2002. Satogaeri only made a booklet appearance in Nikoli Vol. 99, Vol. 100 and Vol. 101. However it got revived on nikoli.com in 2013.

## 2.1. Rules

The official rules from Nikoi state:

1. The areas enclosed by bold lines, are called "Countries". Move the circles, vertically or horizontally, so each country contains only one circle.

2. The numbers in the circles indicate how many cells they have to pass through. Circles without numbers may move any distance, but some of them do not move.

3. The circles cannot cross the tracks of other circles and cannot go over other circles.

And additionally the 4th rule, which counts for every logic puzzle:

  4 A puzzle only has one solution.

    Handcrafted puzzles tend to have a mirrored country-layout either vertically, horizontally or both as can be seen in Figure 2.1.
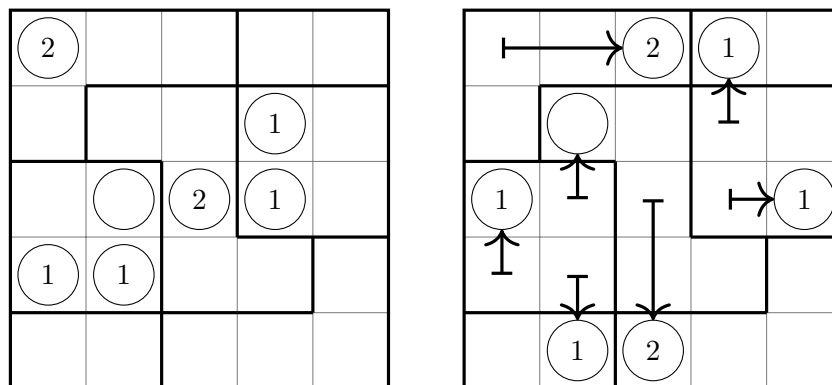


Figure 2.1.: Small sample puzzle with its solution

# 3. Satisfiability Modulo Theories

Before we can go on and talk about how to solve Satogaeri problems we first have to look into Satisfiability Modulo Theories (SMT) for this was used to create the solver and the generator.

SMT can be seen as an enhancement of traditional Satisfiability (SAT) solving. In SAT solving one tries to find an interpretation of a given problem, where this interpretation is a boolean formula composed of boolean variables and expressions like AND, OR and NOT; to state a few. If this constructed formula has a configuration of its variables where the formula evaluates to 'true', then also the original problem must have a solution. A satisfying configuration of the variables often gives a good clue on how to solve the original problem.

To state an example: the formula

$$(a \vee b \vee (\neg c)) \iff ((\neg a) \wedge c)$$

is satisfied with the assignment $a$ is 'false', $b$ and $c$ are 'true'. Therefore the problem interpreted by this formula must be solvable too.

SMT goes one step beyond that by replacing some boolean variables with predicates. A predicate is basically a binary-valued function of non-binary variables, which allows us to have function symbols with different arities. Now we can express formulas in first-order logic like

$$x > y \wedge 3x + 4 \leq 4y$$

where a natural number can be seen as a function symbol with arity 0, a so called constant. There are many different approaches of implementing SMT solvers, but the two major ones are the *eager* and the *lazy* approach.

## 3.1. Eager SMT

In the *eager* approach one tries to convert the given formula into an equisatisfiable propositional formula with all the additional needed constraints of the originating problem. With these constraints the search-space should be reduced so that a common SAT solver can solve the formula much quicker. And this is the appealing part of this approach: one can use already existing SAT solver to finally check the satisfiability. However the converting process into a propositional formula can be cost-intensive. [**?** ]

## 3.2. Lazy SMT

The *lazy* approach consists of two parts. The first part are theory-specific solvers ($\mathcal{T}$-solvers) to handle conjunctions of literals, which are embedded again
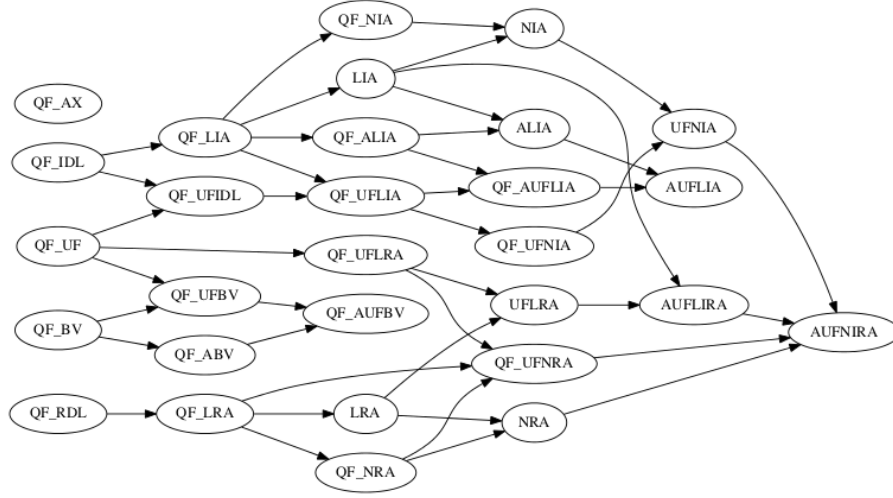
Figure 3.1.: A link from a logic L1 to a logic L2 means that every formula of L1 is also a formula of L2

into a SAT solver. The advantage of incorporating $\mathcal{T}$-solvers is that one can use a specific algorithm and data-structure which fits the problem at hand best. This should lead to a better performance. [**?** ]

## 3.3. SMT-Library

The SMT library (SMT-Lib) was found in 2003 to provide common standards and a library of benchmarks for diverse SMT solvers. With this standards and benchmarks evaluation and comparison of SMT systems should be made easier to give the research a competitive touch, which ultimately should result in advancing the state of the art in SMT. They even organize an annually SMT solver competition (SMT-COMP).

SMT-Lib was greatly useful for this thesis by providing an easily understandable syntax which made many SMT solvers accessible, like Boolector, CVC4 and MathSAT 5 to name a few. On their web-site[1] one can find a long list of solvers supporting SMT-LIB and a separate list of these solvers still under active development.

## 3.4. Logics

SMT-LIB supports a huge variety of different logics. The advantage here is to be able to apply more specialized and therefore more efficient algorithms. The meaning of the logics in Figure 3.1 is letter coded. For example:

1. QF stands for the restriction to quantifier free formulas

2. IA stands for the theory Ints (Integer Arithmetic)

---

[1]www.SMT-LIB.org

3. L before IA, RA, or IRA stands for the linear fragment of those logics

So QF_LIA is unquantified linear integer arithmetic. In essence, Boolean combinations of inequations between linear polynomials over integer variables. Beside this logic we will also use the QF_UF logic throughout this project. QF_UF is the logic of Quantifier-Free Uninterpreted Functions. It incorporates just the Core theory, providing the Bool sort and the various standard operations on Boolean values.

The whole explanation can be found on the SMT-Lib web-site[2] in the section Logics.

## 3.5. CVC4

The SMT solver used in this project is the Cooperating Validity Checker 4 (CVC4), currently the most novel successor of the Stanford Validity Checker which was found in 1996. CVC4 is a joined project of the New York University and the University of Iowa.

Supported features are including: several built-in base theories, support of quantifiers and a model generation ability. The latter is a very important requirement for this project at hand.

Furthermore CVC4 has its own wiki[3] with useful tutorials, a user manual and a developer section.

---

[2]www.SMT-LIB.org
[3]http://cvc4.cs.nyu.edu/wiki

# 4. Solver

Now we will discuss how to represent a Satogaeri puzzle in SMT and what are the required inputs for the CVC4 solver. Because we use the SMT-Lib and its generalized input syntax the CVC4 solver could be replaced by any other solver supporting the SMT-Lib. For illustration purposes we will now manually create a SMT representation of the Satogaeri puzzle of Figure 4.1.

## 4.1. Presetting

Before starting up the CVC4 solver we need to ensure some presetting which is mandatory for our purpose. The parameter additionally given to the program via the command-line are the following:

––lang smt –m ––statistics

   *–lang smt* enables the use of the SMT-Lib syntax other than the regular CVC4 syntax.

   *-m* enables the generation of a model which means once our problem is satisfiable the solver also generates a satisfying assignment for all variables to give us an example model in which the problem has a solution.

   *–statistics* prints a long list of statistics if our problem was satisfiable. We will use these statistics to decide how 'difficult' the given Satogaeri puzzle was. More about that later in Section 8.

## 4.2. Declaring Functions

Now that CVC4 is started with the proper presetting we can start our representation of the puzzle in Figure 4.1.

Figure 4.1.: Trivial Satogaeri puzzle

The first thing to do is to define the logic to be used:

```
( set −logic  QF_LIA)
```

In our case it is unquantified linear integer arithmetic. Simple because we do not need quantifiers nor anything other than simple integer functions. By restricting the search-space for the solver it should be faster in finding a solution.

Finally we can start our SMT-interpretation of the Satogaeri puzzle. First of all we define every cell of the puzzle as a variable integer. One can think of the puzzle as a 3-by-3 matrix filled with yet unknown numbers.

```
( declare −fun  f1 −1  ()  Int )
( declare −fun  f1 −2  ()  Int )
( declare −fun  f1 −3  ()  Int )
( declare −fun  f2 −1  ()  Int )
( declare −fun  f2 −2  ()  Int )
( declare −fun  f2 −3  ()  Int )
( declare −fun  f3 −1  ()  Int )
( declare −fun  f3 −2  ()  Int )
( declare −fun  f3 −3  ()  Int )
```

This lines introduce a new function with the name e.g. 'f2-3' which has no arguments (empty brackets) and returns an integer. 'f2-3' in our case represents the field in row 2 and column 3 (top to bottom, left to right). The collection of these functions will be our matrix, to be more precise its the matrix where we will define our 'country' restrictions of the Satogaeri rules, referred to as country-matrix from here on.

Now we will create a second 3-by-3 matrix just to model the restrictions of the circles.

```
( declare −fun  c1 −1  ()  Int )
( declare −fun  c1 −2  ()  Int )
( declare −fun  c1 −3  ()  Int )
( declare −fun  c2 −1  ()  Int )
( declare −fun  c2 −2  ()  Int )
( declare −fun  c2 −3  ()  Int )
( declare −fun  c3 −1  ()  Int )
( declare −fun  c3 −2  ()  Int )
( declare −fun  c3 −3  ()  Int )
```

Same principle as before only that the 'c' in 'c2-3' stands for circle.

One can think of it as a circle-matrix on top of a country-matrix and both only check for specific rules of the Satogaeri puzzle.

## 4.3. Country Rules

We declared a country-matrix but there are no restrictions on it whatsoever. Next we address the encoding of the rule: 'one or several fields combined result in a country. Every country only can be occupied by one circle.'

For example 'f1-2' and 'f1-3' combined are one country. That means once 'f1-2' is occupied 'f1-3' is not allowed to accommodate any circles and vice versa. Later on we will define that a circle is a positive natural number, whereas the absence of a circle is modelled as '0'. In the SMT syntax it will look like

```
(assert (or (and (> f1-2 0) (= f1-3 0))
            (and (< f1-2 0) (= f1-3 0))))
```

Important to state is that the SMT syntax consists only of prefix notations.

*assert* will make sure that in the end there is an assignment to these fields that this line will become 'true' otherwise the problem is unsatisfiable. The *or* states that at least one of its arguments must be 'true', and the *and* only yields 'true' if all of its arguments evaluate to 'true'. Both *or* and *and* can take one or more arguments inside its brackets whereas *assert* only takes one.

This line basically translates to: either f1-2 is a positive number and f1-3 is zero or vice versa.

Now we have successfully translated rule number 1 into SMT-syntax for one country. Let's do the same with the other two countries:

```
(assert (or
        (and (> f1-1 0) (= f1-2 0) (= f1-3 0) (= f2-3 0))
        (and (= f1-1 0) (> f1-2 0) (= f1-3 0) (= f2-3 0))
        (and (= f1-1 0) (= f1-2 0) (> f1-3 0) (= f2-3 0))
        (and (= f1-1 0) (= f1-2 0) (= f1-3 0) (> f2-3 0))))

(assert (or
        (and (> f2-2 0) (= f3-2 0) (= f3-3 0))
        (and (= f2-2 0) (> f3-2 0) (= f3-3 0))
        (and (= f2-2 0) (= f3-2 0) (> f3-3 0))))
```

And we are done with the country-matrix.

## 4.4. Circle Rules

The first thing we do is to assign every circle with its own ID.

```
(assert (and (= c1-1 1) (= c1-3 2) (= c3-1 3)))
```

At the start of this section we defined c1-2 as a function with no arguments which yields an integer. All we did was to define that this integer has to be 1. In other words: c1-2 is constant 1. Now the 'empty'-circle at c1-1 has the ID 1, '0'-circle at c1-3 has the ID 2 and so on.

Our next step is to define the possible moves every circle is able to make. Let's start of with the easiest: the '0'-circle at c1-3. The 0 indicates that this circle is not allowed to move at all which means he will occupy the field he is already in.

```
(assert (= f1-3 c1-3)))
```

We have our first interaction with the country-matrix and the circle-matrix. With this line we declare that function f1-3 has to yield the same value as c1-3

which is its ID: 2. This leads to, as we remember, that the country consisting of f1-1, f1-2, f1-2 and f2-3 is occupied and every field in this country except f1-3 has to be zero, in other words is not allowed to be occupied by any circle.

Next is the '2'-circle in c1-3. This circle can move to 2 positions either to c1-1 or c3-3 because of its movement-range of 2.

```
( assert ( or
        ( and (= c3−2 c3−1) (= c3−3 c3−1) (= f3−3 c3−1))
        ( and (= c2−1 c3−1) (= c1−1 c3−1) (= f1−1 c3−1))))
```

In this case we not only override the position in the country-matrix but also all the fields we pass in the circle-matrix, because rule 3 states: 'The circles cannot cross the tracks of other circles and cannot go over other circles.' By leaving the ID of every circle in the fields they passed we basically leave a track which is not allowed to be crossed.

For example: the '2'-circle in c1-3 is not allowed to move to c1-1 because 'empty'-circle is already there. We made sure of this by first defining the constant c1-1 with the ID 1 and now we said if the '2'-circle wants to occupy f1-1 it has to leave its ID in c1-1, which yields a contradiction because a constant can never be both 1 and 3.

To make a long story short: in this case only f3-3 can be reached and occupied without violating any rules.

Last one is the 'empty'-circle at c1-1. Because of rule 2: 'Circles without numbers may move any distance, but some of them do not move.' this circle has 5 possible moves to make. Either stay at c1-1, move to c1-2, move to c1-3, move to c2-1 or move to c3-1.

It is worth mentioning that the final solver is smart enough to detect simple collisions like if there is a circle in the path beforehand and will not add that particular move to the SMT encoding, this reduces the possible cases the CVC4 solver has to check.

In this case moving to c1-3 and moving to c3-1 is not an option

```
( assert ( or ( and (= f1−1 c1−1))
            ( and (= c1−2 c1−1) (= f1−2 c1−1))
            ( and (= c2−1 c1−1) (= f2−1 c1−1))))
```

Now we are done. The Satogaeri puzzle of Figure 4.1 is translated into SMT.

## 4.5. Solution

Once we translated the puzzle into an SMT-problem we can evoke the solving procedure by simply typing

```
( check−sat )
```

Now there are 3 possible outputs the solver can give:

1. unsat: It means that there is no possible assignment to solve the problem at hand.

2. unknown: Is used when a solver did not manage to establish whether the set of assertions is satisfiable or not. However some solver will create a candidate model.

3. sat: There is at least one possible assignment which satisfies the problem.

Obviously we looking for a sat result. Once the problem is sat and we enabled *-m*, the model generation, we now can check for possible assignments of all fields in the country-matrix as well as in the circle-matrix. Our major interest lies in the country-matrix:

```
(get−value (f1−1))
(get−value (f1−2))
(get−value (f1−3))
(get−value (f2−1))
(get−value (f2−2))
(get−value (f2−3))
(get−value (f3−1))
(get−value (f3−2))
(get−value (f3−3))
```

Fields that are not occupied by any circle will yield a zero. However the occupied once will give us the ID-number of the circle which is currently occupying this field, the whole country.

We could also look into the circle-matrix with these lines:

```
(get−value (c1−1))
...
(get−value (c3−3))
```

However there we will only see the path taken of every circle marked with its ID and the unneeded fields will be filled with random numbers most likely 0, but it could also be a number used as an ID for a circle which makes the circle-matrix inaccurate.

The output of the solver in this case looks like:

```
((f1−1 0))
((f2−1 0))
((f3−1 3))
((f1−2 1))
((f2−2 0))
((f3−2 0))
((f1−3 0))
((f2−3 0))
((f3−3 2))

((c1−1 1))
((c2−1 0))
((c3−1 3))
((c1−2 1))
```

```
((c2−2 0))
((c3−2 0))
((c1−3 2))
((c2−3 2))
((c3−3 2))
```

## 4.6. Quantifier-Free Uninterpreted Functions

In the hope of speeding up the solving process the logic of Quantifier-Free Uninterpreted Functions was tested too. This logic provides only a few core functions but all we actually need to model a Satogaeri puzzle are equations and inequations.

Let us look into the SMT encoding of this logic. First we enable this particular logic

```
(set−logic QF_UF)
```

Similar to the QF_LIA logic we will build up 2 matrices with variables but this time Integers are not included in our repertory. So we have to introduce a new sort symbol via

```
(declare−sort A 0)
```

This introduces a new sort called A which takes 0 parameters. The next step is to define the variables of this sort A.

```
(declare−fun f1−1 () A)
...
(declare−fun f3−3 () A)

(declare−fun c1−1 () A)
...
(declare−fun c3−3 () A)
```

In the QF_LIA we would have now stated that if a field is occupied by a circle the rest of the country has to be zero. Now that we do not have any numbers we need to define a symbol that represents this zero.

```
(declare−fun zero () A)
```

To keep a link to the previous SMT encoding we simply name this new symbol zero.

Now we model the country restriction. Last time we said: 'If a field is greater than zero the rest has to be zero.' Which could also be stated as: 'If a field is unequal to zero the rest has to be equal to zero."

For example:

```
(assert (or (and (distinct f2−1 zero) (= f3−1 zero))
            (and (= f2−1 zero) (distinct f3−1 zero))))
```

Our next step will be to distinguish all circles and zero from each other. Last time we did that via IDs but this time we simply use distinct:

```
( a s s e r t  ( d i s t i n c t  c1−1 c1−3 c3−1 zero ))
```

All that is left is the circle trace rule which is encoded exactly like what we did with the QF_LIA logic. We use the 'empty'-circle as an example:

```
( a s s e r t  ( or  (=  f1−1 c1−1)
          ( and  (=  c1−2 c1−1)  (=  f1−2 c1−1))
          ( and  (=  c2−1 c1−1)  (=  f2−1 c1−1))))
```

The *(check-sat)* and the *(get-value ())* command work exactly like before. However the output looks a bit different:

```
(( f1−1  @uc_A_0 ))
(( f1−2  @uc_A_0 ))
(( f1−3  @uc_A_2 ))
. . .
```

The difference between the fields is shown via numbers once again, but this time we did not mark the circles with IDs which means if we want to know which circle is represented by which number we need to consult the circle-matrix and trace the numbers back from the occupied country to the origin of the circle. To hear more about that and to find the full encoding QF_UF please consult the Appendix section.

To see how the QF_UF encoding held up against the QF_LIA encoding time-wise please consult Section 8.
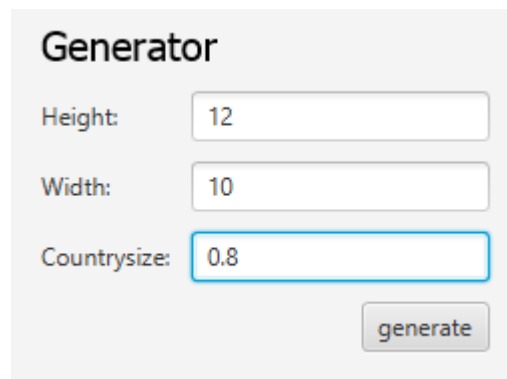
# 5. Generator

The Generator should be able to generate random Satogaeri puzzles with varying size and difficulty. The user enters the width and height of the desired puzzle and another parameter called Countrysize which is a rational number between 0.0 and 1.0 (Figure 5.1). The higher the value of Countrysize the more likely it is to get larger countries. The idea was that the more smaller countries are generated the more difficult the puzzle may get. Unfortunately it contributes very little to the difficulty of the generated puzzle.

## 5.1. Country Generation

As mentioned before the user defines the width and height of the puzzle. Now the program will take the topmost leftmost field which is not yet involved in any country and will declare it as a new country. With the probability of *Countrysize* each of the four connecting fields (top, bottom, left, right) will be added to this newly created country. Next the four connecting fields of each new member have a chance of $(previous\_chance)^2$ to be added too, until there are no new members.

Theoretically with this algorithm it is possible to generate countries in every conceivable shape, however some general shapes are more likely than others. And a very small Countrysize-value will end up in puzzles with a lot of countries consisting only of one field which are too easy.



Figure 5.1.: Generator graphical interface

## 5.2. Circle Generation

After we know the layout of all the countries we can start adding circles. The Generator now randomly picks one field in every country and declares it as the home field of the circle. At this point we do have the solution to the upcoming puzzle. The next step is to shuffle these circles around.

Therefore the Generator randomly picks a direction in which the circle should move and calculates the distance to the next obstacle (e.g.: the boarder of the puzzle, another circle, the track of another circle). Another random number will be picked between 1 and the distance to the next obstacle and this indicates the movement of this circle.

However if the obstacle is right next to the circle a new direction will be chose. If that scenario happens twice in a row, it will become a '0'-circle. This is necessary to reduce the number of boring '0'-circles but do not remove them in general.

Now everything is generated except 'empty'-circles. Those are not be picked randomly but must meet some requirements. A circle will become a 'empty'-circle if and only if:

1. from its home filed on into the moving direction it has to leave its country with the first field

2. in the opposite direction on the origin point there must be a blockade (a circle-trace, a circle, a border or a different country)

3. if it is a '0'-circle it should not be possible to reach another field inside the own country

If one of these requirements is violated it will break the uniqueness of the puzzle.

## 5.3. Verifying Uniqueness

Uniqueness of the solution to a puzzle is very important for the joy of play. In other words the Generator must assure that the puzzle only has one solution. This is done by simply giving the puzzle to the solver with an additional constraint which negates the solution, and if the solver finds a different solution the uniqueness of the puzzle is violated.

But how to model this additional constraint in SMT? To answer that we will look again at the puzzle in Figure 4.1. But this time we need the solution to this puzzle which looks like in Figure 5.2. We remember that for example the 'empty'-circle had the ID 1 and now occupies the field in f2-1. So all we have to do is to tell the home field f2-1 of the 'empty'-circle that the ID 1 is not allowed to land there. And the same goes for the rest of the circles. In the SMT syntax it will look like this:

```
( asser ( or ( not (= f2−1  1))
              ( not (= f1−3  2))
              ( not (= f3−3  3))))
```

Figure 5.2.: Solution to trivial Satogaeri puzzle

If this line is added to our previous example of SMT encoding we have the same SMT problem as before however the solution found in Figure 5.2 is prohibited.

Giving this encoding to the solver will return *unsat* if the given solution is the only one and will return *sat* if there is a second possible solution. Because we enabled the model generating feature we can also have a look at this second solution via the *(get-value ())* input.

In other words after shuffling the numbers the generator checks for a second solution other than the home field positions of the circles, if he finds a different solution he will simply re-shuffle the circles until the home field positions are the unique solution to the puzzle.

# 6. Tool

With the tool one should be able to play crafted and generated Satogaeri puzzles. In addition to that one has the option of drawing puzzles and save them as .pzl files. These files can of course be loaded by the tool so one can share puzzles and play puzzles created by others.

## 6.1. Usability

In Figure 6.1 we can see the main menu and its options.

### 6.1.1. New Game

will open up another menu where on can choose a difficulty for the upcoming puzzle which ranges from Easy, Medium to Hard. After choosing the difficulty one has the option between a few puzzles which were created and published by Nikoli[1].

By dragging the numbers the player can move the circles around, however only vertical and horizontal. Once a number moved into a country this country will turn green to mark the habitation, this feature can be deactivated by the check-box *colored countries*. As can be seen in Figure 6.2 the game also consists of four buttons.

*Clear* returns the origin state of the game, where no circle is moved yet.

*Undo* reverses the last step the user made. (up to 25 steps)

*Check* checks if the puzzle is solved at the current stage.

---

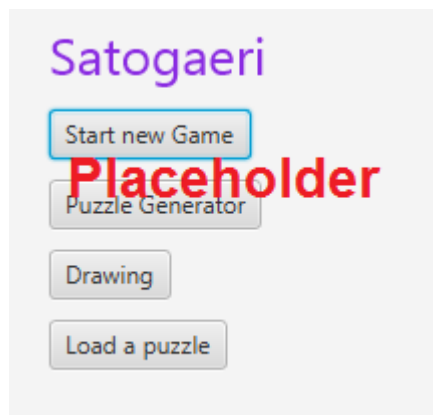[1]http://www.nikoli.com/en/puzzles/satogaeri/
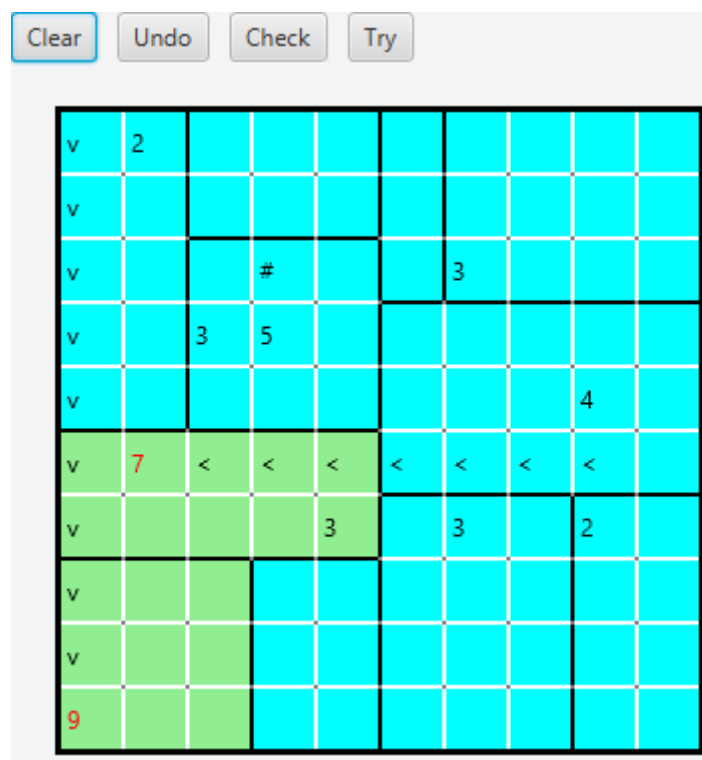


Figure 6.1.: The main menu

Figure 6.2.: The game itself

*Try* enables try mode which basically saves the current state of the puzzle. The *Try* button will be replaced by *Fix* and *Delete* buttons. Fix will disable try mode and keeps all changes the user made during try mode. Delete will also disable try mode, however will return the old, saved state of the puzzle.

### 6.1.2. Generator

will lead to the generator menu which was already shown and explained in Section 5.

### 6.1.3. Drawing

leads to similar menu like in Generator where one can choose the width and height of the puzzle and this is followed by a screen similar to the one in Figure 6.3. Now one can draw one's own puzzle. If in *Draw Country Mode* one can add countries via clicking into a field and hovering over all fields which should be part of this country. By releasing the left mouse button the sequence is over and ever 'added' field will turn green and the country will receive its boarders. Fields that are already in a country can be easily overwritten by hovering over them whilst creating a different country.

In *Draw Circle Mode* one can click on the desired field and a pop-up will arise in which one can enter positive numbers or '#' to enter a circle with the given movement value to this field. If a circle is misplaced one can re-click the field and by typing a negative number the circle will be removed.

Once done one can either click the *save button* to type in the name of the puzzle and create a entered-name.pzl file or click the *solve button* to get a solution to the drawn puzzle. The solver will also state if the solution is unique or not. This is meant to enable a possibility for the user to check if the created puzzle is valid or not. And as an addition to that one can find answers to unsolved Satogaeri puzzles.

### 6.1.4. Loading Puzzle

lets one load a saved .pzl-file to play that puzzle.
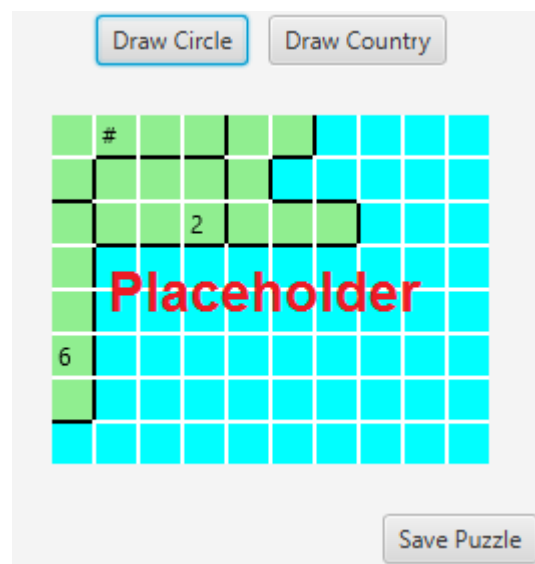
Figure 6.3.: The drawing screen

# 7. JavaFX

The GUI was created with JavaFX because of two simple reasons. First it was appealing to keep the whole project in Java to be operating systems independent and secondly JavaFX is rather young and being up to date is important.

## 7.1. What is JavaFX?

JavaFX was first developed in December 2008 by Oracle Corporation, the current version is JavaFX 8 which was released in May 2014. It is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy client applications that operate consistently across diverse platforms.[1]

It was intended to replace Swing as the standard GUI library in Java Standard Edition however both will be included for now. But still JavaFX can be seen a bit as a successor to Swing.

## 7.2. New Features over Swing

Written as a Java API, JavaFX application code can reference APIs from any Java library. Which means one can use Java API libraries to access native system capabilities. Also the web view supports HTML5 and all the goodies that comes with that.

However for this project the most important part is that JavaFX supports Cascading Style Sheets for skinning ones GUI and FXML to define the GUI separately from the application logic.

FXML files are similar to XML files and they contain all the controls of the GUI. In other words one can finally use the Model-View-Controller pattern without violating some of its regulations.

---

[1]https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm

# 8. Statistics

In this section we will evaluate some benchmarks concerning the solving of Satogaeri puzzles with different logics and by using the 'statistics' option on CVC4 we will find some benchmarks on which we can classify the difficulty of a Satogaeri puzzle.

For the creation of these benchmarks a machine with an Intel(R) Core(TM) i5 CPU M430 prozessor with 4 GB RAM running Windows 8.1 64 bit was used.

## 8.1. Classifying the Difficulty

The statistics option of CVC4 produces a long list of statistics, the full list can be found in the Appendix section. However only a few statistics will help categorizing the difficulty and most of them grow rather with the puzzle size than with the actual difficulty of the puzzle.

The only two numbers we take in account are the 'nonTrivialSatChecks' and the 'sat::conflicts'. Both of them are not influenced by the size of the puzzle, however a single conflict is worth more than a single nonTrivialSatCheck in respect to the difficulty. To get an approximate difficulty level in numbers we add four times the Conflicts to the NonTrivialSatChecks. With this number we now can rank different Satogaeri puzzles.

## 8.2. QF UF vs QF LIA

The time unit in the table below is Second.

We can see that for trivial puzzles the QF_UF logic is faster, however the more difficult it gets the better the QF_LIA logic becomes. This phenomenon

|  | Sakamoto | Aquablue | Alkaline | Casty | Z_H |
|---|---|---|---|---|---|
| Difficulty | Easy | Easy | Medium | Medium | Medium |
| NonTrivial | 9 | 49 | 73 | 84 | 89 |
| Conflicts | 0 | 0 | 0 | 0 | 2 |
| Rating | 9 | 49 | 73 | 84 | 93 |
|  | Cubic Function | -4 | Aiko | T.Karino | bay wolf JR |
| Difficulty | Medium | Hard | Hard | Hard | Hard |
| NonTrivial | 174 | 513 | 581 | 438 | 1137 |
| Conflicts | 5 | 11 | 17 | 59 | 55 |
| Rating | 194 | 557 | 649 | 674 | 1357 |

Figure 8.1.: Table of rated Satogaeri puzzles

|  | Sakamoto | Aquablue | Alkaline | Casty | Z_H |
|---|---|---|---|---|---|
| Difficulty | Easy | Easy | Medium | Medium | Medium |
| Size | 10x10 | 10x10 | 18x10 | 18x10 | 10x10 |
| Rating | 9 | 49 | 73 | 84 | 93 |
| QF_UF | 0.09280818 | 0.1097706 | 0.22391504 | 0.24529236 | 0.13142188 |
| QF_LIA | 0.1312121 | 0.1678718 | 0.28321594 | 0.31705216 | 0.1864898 |
|  | Cubic Function | -4 | Aiko | T.Karino | bay wolf JR |
| Difficulty | Medium | Hard | Hard | Hard | Hard |
| Size | 18x10 | 24x14 | 36x20 | 24x14 | 36x20 |
| Rating | 194 | 557 | 649 | 674 | 1357 |
| QF_UF | 0.27157508 | 0.5942921 | 3.21634508 | 1.00228996 | 3.42382674 |
| QF_LIA | 0.3991329 | 0.69549096 | 1.35633634 | 0.90596628 | 1.72413418 |

Figure 8.2.: Time comparison between QF_UF and QF_LIA

might be explained by the fact that the linear arithmetic finds more use cases in the field and therefore is more optimized, but that is just speculation.

## 8.3. Generating

The generation process includes many random factors and therefore it is impossible to create a representative table with timings. Generating 5x5 puzzles should only take a couple of seconds, however the generation time of a 10x10 puzzle can vary from a couple of seconds to several minutes.

Trying to generate puzzles greater than 12x12 will take a pretty long time (up to 30min), one might get lucky and produce a puzzle with an unique solution within seconds but the waiting time of several minutes is way more likely.

# 9. Conclusion

## 9.1. Solver

The time needed to solve Satogaeri puzzles with this SMT encoding is satisfying. Even the hardest tested puzzle can be solved within 2 seconds. Additionally it is just as fast to find out if a puzzle has an unique solution or not.

## 9.2. Generator

It is possible to generate various puzzles with unique solutions, however exceeding a certain size of a puzzle will end in an inadequate waiting time. One can produce many small puzzles using this generator but for greater ones it is simply to slow.

The reason why it takes so long to generate these puzzles is that once the countries are build the generator shuffles the circles on good luck. If it has more than one solution he will just reshuffle them. A different approach would be finding out which 'area' of the puzzle produces more than one solution and only manipulating these parts.

## 9.3. Tool

The tool incorporates solver and generator as required and additionally provides a drawing feature which should make crafting one owns puzzles easier, especially with the possibility to check for uniqueness of the solution.

The game incorporates the same features as the tool created by Nikoli[1] and additionally to that one can extract the SMT encoding to feed it to other solvers and it is possible to save the puzzle at any moment and continue playing another time.

---

[1]http://www.nikoli.com/en/puzzles/satogaeri/

# A. Exemplary SMT code in QF LIA

```
(set-logic QF_LIA)
(declare-fun f1-1 () Int)
(declare-fun c1-1 () Int)
(declare-fun f1-2 () Int)
(declare-fun c1-2 () Int)
(declare-fun f1-3 () Int)
(declare-fun c1-3 () Int)
(declare-fun f2-1 () Int)
(declare-fun c2-1 () Int)
(declare-fun f2-2 () Int)
(declare-fun c2-2 () Int)
(declare-fun f2-3 () Int)
(declare-fun c2-3 () Int)
(declare-fun f3-1 () Int)
(declare-fun c3-1 () Int)
(declare-fun f3-2 () Int)
(declare-fun c3-2 () Int)
(declare-fun f3-3 () Int)
(declare-fun c3-3 () Int)

(assert (or
(and (> f1-1 0) (= f1-2 0) (= f1-3 0) (= f2-3 0))
(and (= f1-1 0) (> f1-2 0) (= f1-3 0) (= f2-3 0))
(and (= f1-1 0) (= f1-2 0) (> f1-3 0) (= f2-3 0))
(and (= f1-1 0) (= f1-2 0) (= f1-3 0) (> f2-3 0))))

(assert (or (and (> f2-1 0) (= f3-1 0))
(and (= f2-1 0) (> f3-1 0))))

(assert (or (and (> f2-2 0) (= f3-2 0) (= f3-3 0))
(and (= f2-2 0) (> f3-2 0) (= f3-3 0))
(and (= f2-2 0) (= f3-2 0) (> f3-3 0))))

(assert (and (= c1-1 1) (= c1-3 3) (= c3-1 2)))

(assert (or (= f1-1 c1-1)
(and (= c1-2 c1-1) (= f1-2 c1-1))
(and (= c2-1 c1-1) (= f2-1 c1-1))))

(assert (or (= f1-3 c1-3)))
```

```
( assert ( or ( and (= c3−2 c3−1) (= c3−3 c3−1) (= f3−3 c3−1)))))
( check−sat )
( get−value ( f1 −1))
( get−value ( f1 −2))
( get−value ( f1 −3))
( get−value ( f2 −1))
( get−value ( f2 −2))
( get−value ( f2 −3))
( get−value ( f3 −1))
( get−value ( f3 −2))
( get−value ( f3 −3))

Output :
(( f1 −1  0))
(( f1 −2  0))
(( f1 −3  3))
(( f2 −1  1))
(( f2 −2  0))
(( f2 −3  0))
(( f3 −1  0))
(( f3 −2  0))
(( f3 −3  2))
```

# B. Exemplary SMT code in QF UF

```
(set-logic QF_UF)
(declare-sort A 0)
(declare-fun zero () A)
(declare-fun f1-1 () A)
(declare-fun c1-1 () A)
(declare-fun f1-2 () A)
(declare-fun c1-2 () A)
(declare-fun f1-3 () A)
(declare-fun c1-3 () A)
(declare-fun f2-1 () A)
(declare-fun c2-1 () A)
(declare-fun f2-2 () A)
(declare-fun c2-2 () A)
(declare-fun f2-3 () A)
(declare-fun c2-3 () A)
(declare-fun f3-1 () A)
(declare-fun c3-1 () A)
(declare-fun f3-2 () A)
(declare-fun c3-2 () A)
(declare-fun f3-3 () A)
(declare-fun c3-3 () A)

(assert (or
(and (distinct f1-1 zero) (= f1-2 zero) (= f1-3 zero) (= f2-3 zero))
(and (= f1-1 zero) (distinct f1-2 zero) (= f1-3 zero) (= f2-3 zero))
(and (= f1-1 zero) (= f1-2 zero) (distinct f1-3 zero) (= f2-3 zero))
(and (= f1-1 zero) (= f1-2 zero) (= f1-3 zero) (distinct f2-3 zero))))

(assert (or (and (distinct f2-1 zero) (= f3-1 zero))
(and (= f2-1 zero) (distinct f3-1 zero))))

(assert (or (and (distinct f2-2 zero) (= f3-2 zero) (= f3-3 zero))
(and (= f2-2 zero) (distinct f3-2 zero) (= f3-3 zero))
(and (= f2-2 zero) (= f3-2 zero) (distinct f3-3 zero))))

(assert (distinct c1-1 c1-3 c3-1 zero))

(assert (or (= f1-1 c1-1)
(and (= c1-2 c1-1) (= f1-2 c1-1))
(and (= c2-1 c1-1) (= f2-1 c1-1))))
```

```
( assert ( or (= f1−3 c1−3)))

( assert ( or (and (= c3−2 c3−1) (= c3−3 c3−1) (= f3−3 c3−1))))
( check−sat )
( get−value ( f1 −1))
( get−value ( f1 −2))
( get−value ( f1 −3))
( get−value ( f2 −1))
( get−value ( f2 −2))
( get−value ( f2 −3))
( get−value ( f3 −1))
( get−value ( f3 −2))
( get−value ( f3 −3))

( get−value ( c1 −1))
( get−value ( c1 −2))
( get−value ( c1 −3))
( get−value ( c2 −1))
( get−value ( c2 −2))
( get−value ( c2 −3))
( get−value ( c3 −1))
( get−value ( c3 −2))
( get−value ( c3 −3))

Output :
(( f1 −1  @uc_A_0 ))
(( f1 −2  @uc_A_0 ))
(( f1 −3  @uc_A_2 ))
(( f2 −1  @uc_A_1 ))
(( f2 −2  @uc_A_0 ))
(( f2 −3  @uc_A_0 ))
(( f3 −1  @uc_A_0 ))
(( f3 −2  @uc_A_0 ))
(( f3 −3  @uc_A_3 ))

(( c1 −1  @uc_A_1 ))
(( c1 −2  @uc_A_1 ))
(( c1 −3  @uc_A_2 ))
(( c2 −1  @uc_A_1 ))
(( c2 −2  @uc_A_0 ))
(( c2 −3  @uc_A_0 ))
(( c3 −1  @uc_A_3 ))
(( c3 −2  @uc_A_3 ))
(( c3 −3  @uc_A_3 ))
```

We know that in f3-3 the circle with the 'ID' 3 landed. in the circle-matrix

we can trace this 'ID' back via c3-3, c3-2 to c3-1. In other words the circle which occupied f3-3 had it's origin in c3-1, which is the '2'-circle.

One can also observe that c1-1, c1-2 and c2-1 share the same 'ID' which makes little sense, for a circle cannot move in curves. This is because c2-1 actually never was visited by the '#'-circle but the solver will give unused fields in the circle-matrix 'IDs' at random. Like stated earlier in most cases it will be '0' however there is a chance that it is a different number.

However this is not a real problem, for as long as one tracks from the occupied field back to the origin one will always determine which circle belongs to which country.

# C. Statistics Output

This is the full output of the CVC4 solver after solving the 'bay wolf JR' puzzle.

```
expr :: ExprManager :: AND,  1359
expr :: ExprManager :: EQUAL,  8817
expr :: ExprManager :: GT,  720
expr :: ExprManager :: OR,  193
expr :: ExprManager :: SEXPR,  720
expr :: ExprManager :: VARIABLE: Integer  type ,  1440
DefaultModel :: constantTermsCount ,  0
DefaultModel :: functionTermsCount ,  0
DefaultModel :: mergesCount ,  1263
DefaultModel :: termsCount ,  1360
FirstOrderModelIG :: constantTermsCount ,  0
FirstOrderModelIG :: functionTermsCount ,  0
FirstOrderModelIG :: mergesCount ,  0
FirstOrderModelIG :: termsCount ,  2
InstantiationEngine :: Instantiations_Auto_Gen ,  0
InstantiationEngine :: Instantiations_Auto_Gen_Min ,  0
InstantiationEngine :: Instantiations_Cbqi_Arith ,  0
InstantiationEngine :: Instantiations_Cbqi_Arith_Minus ,  0
InstantiationEngine :: Instantiations_Cbqi_Datatypes ,  0
InstantiationEngine :: Instantiations_Guess ,  0
InstantiationEngine :: Instantiations_User_Patterns ,  0
QuantifiersEngine :: Duplicate_Inst ,  0
QuantifiersEngine :: Duplicate_Inst_Eq ,  0
QuantifiersEngine :: Instantiations_Total ,  0
QuantifiersEngine :: MonoCandidatesCacheHit ,  0
QuantifiersEngine :: MonoCandidatesCacheMiss ,  0
QuantifiersEngine :: MultiCandidatesCacheHit ,  0
QuantifiersEngine :: MultiCandidatesCacheMiss ,  0
QuantifiersEngine :: Multi_Trigger_Instantiations ,  0
QuantifiersEngine :: NumMonoCandidates ,  0
QuantifiersEngine :: NumMonoCandidatesNewTerm ,  0
QuantifiersEngine :: NumMultiCandidates ,  0
QuantifiersEngine :: Num_Quantifiers ,  0
QuantifiersEngine :: Rounds_Instantiation_Full ,  0
QuantifiersEngine :: Rounds_Instantiation_Last_Call ,  0
QuantifiersEngine :: Term_in_TermDb ,  0
QuantifiersEngine :: Triggers ,  0
QuantifiersEngine :: Triggers_Multi ,  0
QuantifiersEngine :: Triggers_Simple ,  0
```

```
QuantifiersEngine :: lit_phase_nreq , 0
QuantifiersEngine :: lit_phase_req , 0
SharedTermsDatabase :: constantTermsCount , 0
SharedTermsDatabase :: functionTermsCount , 0
SharedTermsDatabase :: mergesCount , 0
SharedTermsDatabase :: termsCount , 2
TheoryEngine :: combineTheoriesTime , 0.000000
TheoryRewriteRules :: CacheHit , 0
TheoryRewriteRules :: CacheMiss , 0
TheoryRewriteRules :: Check , 0
TheoryRewriteRules :: FullCheck , 0
TheoryRewriteRules :: MatchFound , 0
TheoryRewriteRules :: Num_RewriteRules , 0
TheoryRewriteRules :: Poll , 0
pivots :: sat , "[]"
pivots :: unkown , "[]"
pivots :: unsat , "[]"
preprocessor :: number of unconstrained elims , 0
prop :: theoryproxy :: replayedDecisions , 0
sat :: clauses_literals , 50140
sat :: conflicts , 55
sat :: decisions , 1341
sat :: learnts_literals , 242
sat :: max_literals , 263
sat :: propagations , 33499
sat :: rnd_decisions , 0
sat :: starts , 3
sat :: tot_literals , 262
smt :: SmtEngine :: checkModelTime , 0.000000
smt :: SmtEngine :: cnfConversionTime , 0.211713
smt :: SmtEngine :: definitionExpansionTime , 0.180142
smt :: SmtEngine :: iteRemovalTime , 0.812807
smt :: SmtEngine :: miplibPassTime , 0.000000
smt :: SmtEngine :: nonclausalSimplificationTime , 0.154182
smt :: SmtEngine :: numAssertionsPostITERemoval , 202
smt :: SmtEngine :: numAssertionsPreITERemoval , 202
smt :: SmtEngine :: numConstantProps , 0
smt :: SmtEngine :: numMiplibAssertionsRemoved , 0
smt :: SmtEngine :: processAssertionsTime , 1.587684
smt :: SmtEngine :: pushPopTime , 0.000000
smt :: SmtEngine :: rewriteBooleanTermsTime , 0.100083
smt :: SmtEngine :: simpITETime , 0.030055
smt :: SmtEngine :: simplifiedToFalse , 0
smt :: SmtEngine :: solveTime , 0.259811
smt :: SmtEngine :: staticLearningTime , 0.061121
smt :: SmtEngine :: theoryPreprocessTime , 0.415322
smt :: SmtEngine :: unconstrainedSimpTime , 0.000000
```

```
theory :: QuantifiersEngine :: time ,  0.000000
theory :: arith :: ArithCongruenceManager :: constantTermsCount ,  0
theory :: arith :: ArithCongruenceManager :: functionTermsCount ,  0
theory :: arith :: ArithCongruenceManager :: mergesCount ,  4405
theory :: arith :: ArithCongruenceManager :: termsCount ,  4826
theory :: arith :: AssertLowerConflicts ,  0
theory :: arith :: AssertUpperConflicts ,  0
theory :: arith :: DisequalityConflicts ,  0
theory :: arith :: DisequalitySplits ,  0
theory :: arith :: FC :: Construction ,  0.000000
theory :: arith :: FC :: FoundSat ,  0
theory :: arith :: FC :: FoundUnsat ,  0
theory :: arith :: FC :: Missed ,  0
theory :: arith :: FC :: Timer ,  0.000000
theory :: arith :: FC :: UpdateConflicts ,  0
theory :: arith :: FC :: initialProcessTime ,  0.000000
theory :: arith :: FC :: lastPivots ,  ”0”
theory :: arith :: FC :: selectUpdateForDualLike ,  0.000000
theory :: arith :: FC :: selectUpdateForPrimal ,  0.000000
theory :: arith :: SOI :: ConfMin :: num, 0
theory :: arith :: SOI :: Conflict :: Minimization ,  0.000000
theory :: arith :: SOI :: Construction ,  0.000000
theory :: arith :: SOI :: FoundSat ,  0
theory :: arith :: SOI :: FoundUnsat ,  0
theory :: arith :: SOI :: HasToBeMin ,  0
theory :: arith :: SOI :: MaybeNotMin ,  0
theory :: arith :: SOI :: Missed ,  0
theory :: arith :: SOI :: Time ,  0.000000
theory :: arith :: SOI :: UpdateConflicts ,  0
theory :: arith :: SOI :: initialProcessTime ,  0.000000
theory :: arith :: SOI :: lastPivots ,  ”0”
theory :: arith :: SOI :: selectSOI ,  0.000000
theory :: arith :: SlackVariables ,  0
theory :: arith :: UserVariables ,  1263
theory :: arith :: adjTime ,  0.000000
theory :: arith :: attempt :: conflicts ,  0
theory :: arith :: attempt :: queueTime ,  0.000000
theory :: arith :: attempt :: searchTime ,  0.000000
theory :: arith :: bound :: boundComputations ,  0
theory :: arith :: bound :: boundPropagations ,  0
theory :: arith :: bound :: time ,  0.010008
theory :: arith :: cd :: unatePropagateCalls ,  9631
theory :: arith :: cd :: unatePropagateImplications ,  615
theory :: arith :: congruence :: conflicts ,  0
theory :: arith :: congruence :: equalsConstantCalls ,  4405
theory :: arith :: congruence :: propagateConstraints ,  0
theory :: arith :: congruence :: propagations ,  0
```

```
theory :: arith :: congruence :: watchedVariableIsNotZero ,  0
theory :: arith :: congruence :: watchedVariableIsZero ,  0
theory :: arith :: congruence :: watchedVariables ,  0
theory :: arith :: currSetToSmaller ,  0
theory :: arith :: dio :: conflictCalls ,  0
theory :: arith :: dio :: conflictTimer ,  0.000000
theory :: arith :: dio :: conflicts ,  0
theory :: arith :: dio :: cutCalls ,  0
theory :: arith :: dio :: cutTimer ,  0.000000
theory :: arith :: dio :: cuts ,  0
theory :: arith :: dual :: UpdateConflicts ,  0
theory :: arith :: dual :: findConflictOnTheQueueTime ,  0.000000
theory :: arith :: dual :: lastPivots ,  "0"
theory :: arith :: dual :: recentViolationCatches ,  0
theory :: arith :: dual :: searchTime ,  0.000000
theory :: arith :: dual :: simplexConflicts ,  0
theory :: arith :: externalBranchAndBounds ,  0
theory :: arith :: forcing :: time ,  0.000000
theory :: arith :: initialTableauSize ,  0
theory :: arith :: iteConstantApplications ,  0
theory :: arith :: iteMinMaxApplications ,  0
theory :: arith :: newPropTimer ,  0.069739
theory :: arith :: pivotTime ,  0.000000
theory :: arith :: pivots ,  0
theory :: arith :: ppRewriteTimer ,  0.321978
theory :: arith :: pqueue :: enqueues ,  0
theory :: arith :: pqueue :: enqueuesCollection ,  0
theory :: arith :: pqueue :: enqueuesCollectionDuplicates ,  0
theory :: arith :: pqueue :: enqueuesDiffMode ,  0
theory :: arith :: pqueue :: enqueuesVarOrderMode ,  0
theory :: arith :: pqueue :: enqueuesVarOrderModeDuplicates ,  0
theory :: arith :: presolveTime ,  0.803153
theory :: arith :: restartTimer ,  0.000000
theory :: arith :: simplifyTimer ,  0.009655
theory :: arith :: smallerSetToCurr ,  0
theory :: arith :: staticLearningTimer ,  0.030386
theory :: arith :: status :: avgUnknownsInARow ,  0.000000
theory :: arith :: status :: commitsOnConflicts ,  0
theory :: arith :: status :: maxUnknownsInARow ,  0
theory :: arith :: status :: nontrivialSatChecks ,  1137
theory :: arith :: status :: revertsOnConflicts ,  0
theory :: arith :: status :: unknowns ,  0
theory :: arith :: updates ,  600
theory :: arith :: weakening :: attempts ,  0
theory :: arith :: weakening :: success ,  0
theory :: arith :: weakening :: time ,  0.000000
theory :: arith :: weakening :: total ,  0
```

```
theory :: arrays :: TheoryArrays :: constantTermsCount ,  0
theory :: arrays :: TheoryArrays :: functionTermsCount ,  0
theory :: arrays :: TheoryArrays :: mergesCount ,  0
theory :: arrays :: TheoryArrays :: termsCount ,  2
theory :: arrays :: TheoryArraysMayEqual :: constantTermsCount ,  0
theory :: arrays :: TheoryArraysMayEqual :: functionTermsCount ,  0
theory :: arrays :: TheoryArraysMayEqual :: mergesCount ,  0
theory :: arrays :: TheoryArraysMayEqual :: termsCount ,  2
theory :: arrays :: TheoryArraysPP :: constantTermsCount ,  0
theory :: arrays :: TheoryArraysPP :: functionTermsCount ,  0
theory :: arrays :: TheoryArraysPP :: mergesCount ,  0
theory :: arrays :: TheoryArraysPP :: termsCount ,  2
theory :: arrays :: avgInStoresListLength ,  0.000000
theory :: arrays :: avgIndexListLength ,  0.000000
theory :: arrays :: avgStoresListLength ,  0.000000
theory :: arrays :: callsMergeInfo ,  0
theory :: arrays :: checkTime ,  0.000000
theory :: arrays :: infoTableSize ,  0
theory :: arrays :: listsCount ,  0
theory :: arrays :: maxList ,  0
theory :: arrays :: mergeInfoTimer ,  0.000000
theory :: arrays :: number of Ext lemmas ,  0
theory :: arrays :: number of Row lemmas ,  0
theory :: arrays :: number of  calls to setNonLinear ,  0
theory :: arrays :: number of  explanations ,  0
theory :: arrays :: number of  getModelVal conflicts ,  0
theory :: arrays :: number of  getModelVal splits ,  0
theory :: arrays :: number of  propagations ,  0
theory :: arrays :: number of  setModelVal conflicts ,  0
theory :: arrays :: number of  setModelVal splits ,  0
theory :: arrays :: number of  shared array var splits ,  0
theory :: bv :: AvgBVConflictSize ,  0.000000
theory :: bv :: BitblastSolver :: NumCallsToCheck ,  0
theory :: bv :: BitblastTimer ,  0.000000
theory :: bv :: CoreSolver :: NumCallsToCheck ,  0
theory :: bv :: InequalitySolver :: NumCallsToCheck ,  0
theory :: bv :: NumberOfAtomSatClauses ,  0
theory :: bv :: NumberOfBitblastedAtoms ,  0
theory :: bv :: NumberOfBitblastedTerms ,  0
theory :: bv :: NumberOfFullCheckCalls ,  0
theory :: bv :: NumberOfSolveSubstitutions ,  0
theory :: bv :: NumberOfStandardCheckCalls ,  0
theory :: bv :: NumberOfTermSatClauses ,  0
theory :: bv :: TheoryBV :: constantTermsCount ,  0
theory :: bv :: TheoryBV :: functionTermsCount ,  0
theory :: bv :: TheoryBV :: mergesCount ,  0
theory :: bv :: TheoryBV :: termsCount ,  2
```

```
theory :: bv :: bvminisat :: TotalVariableEliminationTime ,  0.000000
theory :: bv :: bvminisat :: calls_to_solve ,  0
theory :: bv :: bvminisat :: clauses_literals ,  0
theory :: bv :: bvminisat :: conflicts ,  0
theory :: bv :: bvminisat :: decisions ,  0
theory :: bv :: bvminisat :: eliminated_vars ,  0
theory :: bv :: bvminisat :: learnts_literals ,  0
theory :: bv :: bvminisat :: max_literals ,  0
theory :: bv :: bvminisat :: propagations ,  0
theory :: bv :: bvminisat :: rnd_decisions ,  0
theory :: bv :: bvminisat :: solve_time ,  0.000000
theory :: bv :: bvminisat :: starts ,  0
theory :: bv :: bvminisat :: tot_literals ,  0
theory :: bv :: slicer :: AverageFindDepth ,  0.000000
theory :: bv :: slicer :: NumberOfEqualitiesAdded ,  "0"
theory :: bv :: slicer :: NumberOfMerges ,  0
theory :: bv :: slicer :: NumberOfRepresentatives ,  0
theory :: bv :: slicer :: NumberOfSplits ,  0
theory :: bv :: solveTimer ,  0.000000
theory :: bv :: weightComputationTimer ,  0.000000
theory :: datatypes :: TheoryDatatypes :: constantTermsCount ,  0
theory :: datatypes :: TheoryDatatypes :: functionTermsCount ,  0
theory :: datatypes :: TheoryDatatypes :: mergesCount ,  0
theory :: datatypes :: TheoryDatatypes :: termsCount ,  2
theory :: quantifiers :: theoryTime ,  0.000000
theory :: rewriterules :: theoryTime ,  0.000000
theory :: shared_terms ,  0
theory :: strings :: TheoryStrings :: constantTermsCount ,  0
theory :: strings :: TheoryStrings :: functionTermsCount ,  0
theory :: strings :: TheoryStrings :: mergesCount ,  0
theory :: strings :: TheoryStrings :: termsCount ,  2
theory :: uf :: TheoryUF :: constantTermsCount ,  0
theory :: uf :: TheoryUF :: functionTermsCount ,  0
theory :: uf :: TheoryUF :: mergesCount ,  0
theory :: uf :: TheoryUF :: termsCount ,  2
theory :: uf :: symmetry_breaker :: clauses ,  0
theory :: uf :: symmetry_breaker :: permutationSetsConsidered ,  0
theory :: uf :: symmetry_breaker :: permutationSetsInvariant ,  0
theory :: uf :: symmetry_breaker :: timers :: initNormalization ,  0.000000
theory :: uf :: symmetry_breaker :: timers :: invariantByPermutations ,  0.000000
theory :: uf :: symmetry_breaker :: timers :: selectTerms ,  0.000000
theory :: uf :: symmetry_breaker :: units ,  0
theory<THEORY_ARITH>:: computeCareGraphTime ,  0.000000
theory<THEORY_ARITH>:: conflicts ,  0
theory<THEORY_ARITH>:: flipDecision ,  0
theory<THEORY_ARITH>:: lemmas ,  3238
theory<THEORY_ARITH>:: propagations ,  0
```

```
theory<THEORY_ARITH>::requirePhase, 0
theory<THEORY_ARITH>::restartDemands, 0
theory<THEORY_ARRAY>::computeCareGraphTime, 0.000000
theory<THEORY_ARRAY>::conflicts, 0
theory<THEORY_ARRAY>::flipDecision, 0
theory<THEORY_ARRAY>::lemmas, 0
theory<THEORY_ARRAY>::propagations, 0
theory<THEORY_ARRAY>::requirePhase, 0
theory<THEORY_ARRAY>::restartDemands, 0
theory<THEORY_BOOL>::computeCareGraphTime, 0.000000
theory<THEORY_BOOL>::conflicts, 0
theory<THEORY_BOOL>::flipDecision, 0
theory<THEORY_BOOL>::lemmas, 0
theory<THEORY_BOOL>::propagations, 0
theory<THEORY_BOOL>::requirePhase, 0
theory<THEORY_BOOL>::restartDemands, 0
theory<THEORY_BUILTIN>::computeCareGraphTime, 0.000000
theory<THEORY_BUILTIN>::conflicts, 0
theory<THEORY_BUILTIN>::flipDecision, 0
theory<THEORY_BUILTIN>::lemmas, 0
theory<THEORY_BUILTIN>::propagations, 0
theory<THEORY_BUILTIN>::requirePhase, 0
theory<THEORY_BUILTIN>::restartDemands, 0
theory<THEORY_BV>::computeCareGraphTime, 0.000000
theory<THEORY_BV>::conflicts, 0
theory<THEORY_BV>::flipDecision, 0
theory<THEORY_BV>::lemmas, 0
theory<THEORY_BV>::propagations, 0
theory<THEORY_BV>::requirePhase, 0
theory<THEORY_BV>::restartDemands, 0
theory<THEORY_DATATYPES>::computeCareGraphTime, 0.000000
theory<THEORY_DATATYPES>::conflicts, 0
theory<THEORY_DATATYPES>::flipDecision, 0
theory<THEORY_DATATYPES>::lemmas, 0
theory<THEORY_DATATYPES>::propagations, 0
theory<THEORY_DATATYPES>::requirePhase, 0
theory<THEORY_DATATYPES>::restartDemands, 0
theory<THEORY_QUANTIFIERS>::computeCareGraphTime, 0.000000
theory<THEORY_QUANTIFIERS>::conflicts, 0
theory<THEORY_QUANTIFIERS>::flipDecision, 0
theory<THEORY_QUANTIFIERS>::lemmas, 0
theory<THEORY_QUANTIFIERS>::propagations, 0
theory<THEORY_QUANTIFIERS>::requirePhase, 0
theory<THEORY_QUANTIFIERS>::restartDemands, 0
theory<THEORY_REWRITERULES>::computeCareGraphTime, 0.000000
theory<THEORY_REWRITERULES>::conflicts, 0
theory<THEORY_REWRITERULES>::flipDecision, 0
```

```
theory<THEORY_REWRITERULES>::lemmas ,  0
theory<THEORY_REWRITERULES>:: propagations ,  0
theory<THEORY_REWRITERULES>:: requirePhase ,  0
theory<THEORY_REWRITERULES>:: restartDemands ,  0
theory<THEORY_STRINGS>::computeCareGraphTime ,  0.000000
theory<THEORY_STRINGS>:: conflicts ,  0
theory<THEORY_STRINGS>:: flipDecision ,  0
theory<THEORY_STRINGS>::lemmas ,  0
theory<THEORY_STRINGS>:: propagations ,  0
theory<THEORY_STRINGS>:: requirePhase ,  0
theory<THEORY_STRINGS>:: restartDemands ,  0
theory<THEORY_UF>::computeCareGraphTime ,  0.000000
theory<THEORY_UF>:: conflicts ,  0
theory<THEORY_UF>:: flipDecision ,  0
theory<THEORY_UF>::lemmas ,  0
theory<THEORY_UF>:: propagations ,  0
theory<THEORY_UF>:: requirePhase ,  0
theory<THEORY_UF>:: restartDemands ,  0
driver :: filename ,  <stdin>
driver :: sat/unsat ,  sat
driver :: totalTime ,  1.779003200
```